A description of ISIS can be found in [7]. In this paper we describe a fundamental element of a new system called HORUS[1] being built at Cornell. HORUS has evolved from ISIS after much experience with building practical fault-tolerant distributed systems.

This work was motivated by a trend in the use of ISIS process groups that has become apparent over the last eight years. The process group paradigm is popular with ISIS applications programmers; almost every major application written using ISIS makes extensive use of process groups. In their original design, process groups were intended as a coarse grain transport mechanism for communicating with multiple processes. Process groups were used to represent a replicated service. However, the paradigm has proven popular for more fine grain uses. Over the last few years applications written using ISIS have used process groups to represent *objects* rather than services. This trend has impacted the original design in several ways and has lead us to focus our attention on providing *light-weight process groups*.

The architecture of HORUS was influenced by microkernel design concepts, in which several light-weight mechanisms are provided in user space. The most obvious of these is the light-weight process or thread abstraction[8, 15]. Another well-known, older abstraction is memory allocation. These abstractions not only allow easier resource management by sharing most of a core environment, but also provide a portable interface across different environments.

The basic idea behind the light-weight process group (LWG) abstraction is that many LWGs are mapped to a single core group (or set of core groups) as implemented by the kernel of HORUS. Thus, these LWGs share the same security environment (much like threads share the same address space), and the same failure model, while their messages are multiplexed over a single core group transport. The benefit of this approach is that membership changes to the core group automatically affect large numbers of LWGs, amortizing the cost of maintaining membership information over what the application considers a large number of independent groups. The ISIS system lacks such a facility, forcing many application programmers to develop equivalent mechanisms.

We have built a prototype of LWGs on top of ISIS V3.0.6 and the initial results show significant improvements in performance. In particular, the LWG subsystem allows LWGs to share the same failure detection protocol execution thereby resulting in faster reaction to member failures and reduced network load. Execution times for typical group operations are also improved: the initial prototype has a speed-up factor of nine for the group create operation (the resulting speed is about

[1]In Egyptian mythology, HORUS is the son of ISIS.

30 ms), and even higher speed-ups for group joins and leaves.

To motivate the problem, we present several examples of how fine grain process groups help solve problems present in distributed applications. We then briefly present the architecture of the HORUS system with particular attention to the light-weight group subsystem. We follow this with a discussion of the key aspects of light-weight process groups and present the basic portions of an interface to our subsystem. We conclude with some initial performance results.

2 Trends in the use of Process Groups

In this section we look at the use of ISIS process groups in three major applications written on top of the ISIS system. By looking at these and other applications we gained insight into how to improve the performance and functionality of process groups.

2.1 The Deceit File System

Our first example of a practical fault-tolerant distributed application is the Deceit file system [14]. Deceit is an NFS-compatible file system that replicates its files across a collection of servers. The system provides flexible support for fault-tolerance. A set of parameters attached to each file controls its replication level and update semantics. As the system is used, file replicas migrate to form working sets on the servers that are currently receiving requests. Deceit's file system therefore exists as a whole across all of the servers yet no one server need contain the whole file system. A key aspect of Deceit is its ability to maintain one-copy serializability in the event of server failures and distributed requests and updates. To manage the inherent complexity of achieving such a property, Deceit uses an ISIS process group to represent the replicas of a file; each member of the group actively maintains a replica of the file. This set of servers changes dynamically as replicas migrate and as servers crash and recover.

Logically, an update to a file need only be multicast to the collection of servers maintaining replicas of that file using the ISIS process group as the transport mechanism. The initial design of this system was built in the obvious way; a single process group was associated with each file's set of replicas. It became quite apparent however that this was not the correct approach for using ISIS process groups; the system suffered greatly from performance problems. Too many process groups that were created (one for each file in the file system) and the algorithms that provide the ordering semantics of group communication were greatly affected by this (we will discuss this later).

A few observations about the collection of process groups lead us to the design decisions that contribute to the good performance of today's Deceit and to the foundation of light-weight process groups. First, good fault-tolerance was obtained with a relatively small collection of file servers. Three to five servers provide good availability, reliability, and performance. Second, even though many (thousands of) process groups were desired, the number of unique process groups, in terms of their membership, was quite small. By using a single process group for the collection of files that had the same replica set, the number of process groups was dramatically reduced with a corresponding improvement in performance. In this new design, when replicas migrated they needed to change process groups, by orchestrating the change through a coordinator in the group. Deceit was able to use the inexpensive CBCAST protocol [6] while maintaining the consistency of the file's replicas.

2.2 The ISIS Transaction Tool

The ISIS Toolkit includes a tool for distributed transactions [9]. A transaction is represented by a process group comprising all the servers which have an interest in the outcome of the transaction (the participants). The implementation of the tool in ISIS is very straightforward. Reliable group multicast is used to implement the commit protocol, and group monitoring facilities are used to detect the failure of transaction participants and to trigger a transaction abort. To ensure that the state of a transaction persists even when all participants fail, transaction state is logged to disk, and transaction outcomes are logged to the transaction recovery manager, itself implemented by a process group.

While the semantics of ISIS process groups and reliable multicast greatly simplified the implementation of the transaction tool, performance was poor. The transaction tool needed only anonymous groups, but ISIS required every group to have a name. The transaction tool generates a known-to-be-unique name derived from the transaction identifier. ISIS incurs unnecessary costs verifying the name's uniqueness by multicasting to the ISIS servers on the network when the group is created, and searching for the name during subsequent join operations. This deficiency is fixed in HORUS, which directly supports anonymous groups and leaves naming to an external service.

More serious than group naming was the cost of a group join. The critical path of a transaction included one group join for every participant and a single group deletion at transaction end. A join involves synchronizing all the current members of the group, and possibly the authentication of the new member. One common scenario in the use of the transaction tool is for a client to issue a series of transactions to the same set of servers. After each transaction the group is torn down only to be built again by the following transaction. This creates unnecessary work when the group transport could be saved.

2.3 META

META [18, 10] is a system for distributed management. It provides a mechanism for instrumenting programs with sensors and actuators and allows creating sophisticated reactive control systems in a distributed network. META makes use of ISIS for its group communication and fault-tolerance. Process groups in META are used both to maintain *aggregates* and as a convenient naming mechanism. Aggregates are used to represent a collection of machines that satisfy some property (e.g., a set of machines with a light load). This collection is maintained (determined) by a set of replicas which detect changes in the aggregate set. An ISIS process group is used to manage this replica set. Aggregates are a fundamental piece of META and are intended for heavy use by META applications, and consequently, META shows similar characteristics to Deceit: a relatively small set of replicas can be responsible for a large number of coincident process groups. Like the initial design of Deceit, the failure of a replica can trigger a flood of distributed agreement protocol invocations.

3 Analysis of Performance Problems

In general we have found that good performance can be obtained from group communication in ISIS provided that the programmer has solid knowledge of the protocol semantics and knows the details of the implementation well enough to make optimizations. Each of the authors in the above systems are sophisticated ISIS programmers that took the semantics of the ISIS communication system and knowledge of the internal protocols into account when designing their software. In general one cannot expect typical applications programmers to be (or want to be!) as knowledgeable about ISIS as these authors. This has motivated us to consider light-weight process groups as a necessary piece of the HORUS system. LWGs should allow applications programmers to use the process group paradigm in a manner which fits the logical structure of their application and which yields good performance.

We now look at why the original process group mechanism in ISIS performed poorly for these applications. The performance problems are mainly a result of the process group algorithms being too closely coupled with the interface provided to the applications builder. Three major performance problems illustrate this point.

3.1 Failure detection

ISIS provides a strong guarantee of consistency for group membership changes. A group's membership history can be characterized by a total order on the join and leave/failure events of the group. Each group member observes the membership in an order consistent with this history. In addition, ISIS provides the strong guarantee of failure atomicity; messages are delivered in the same view of the group's membership at all correct and functional destinations[2]. This allows the recipients to make efficient local decisions about the global state of the system without the need for extra communication. [13] and [6] present the semantics of ISIS process groups and group communication.

Figure 1 shows an example of communication with and without failure atomicity. Failure atomicity and serialized membership greatly impact the performance of process groups when failures occur. Consider the fault-tolerant NFS file server described above if it made a naive use of process groups (by creating one process group per file). At some point during the normal operation there might be a thousand or more process groups representing the files actively in use that are being maintained on three servers. If one of these servers should fail, the *ISIS* group membership and atomicity protocols would trigger for each of these one thousand groups, forcing failure atomicity on the outstanding messages, delivering them in consistent views across their recipients. Each of these instantiations would force an expensive flush of the group's communication. Unfortunately this would have the disastrous impact of flooding the network with protocol messages, which can lead to congestion and the ultimate "failure" of other processes in the system, causing a "domino" effect.

3.2 Overlapping Groups

ISIS provides strong causality guarantees for group communication. This guarantee applies to communication that spans groups. This is an important property of the ISIS system because it allows for less constrictive communication and allows groups to be used flexibly. [6] discusses the ramifications of this property on the algorithms that must implement it. Currently the ISIS system uses a conservative protocol. In order to send a message m to a group G, G must be the only "active"

[2]In the case of partitions, this atomicity cannot be guaranteed, but the partioned processes will form their own consistent groups within which atomicity is respected.

group. A group G' is active for a process p if there is some message m' to G' that has been transmitted by p or delivered to p and which p considers unstable. A process considers a message stable if it learns that the message has been received at all of its destinations. If there is more than one group active for a process, it must block the transmission of a message m until all other groups become inactive (ie. until their messages become stable). This delay may requi-e waiting for acknowledgements from all members of a previous multicast, and potentially for stability information from other groups. In Figure 2 (a), C must delay its multicast to B and D until it learns that the causally preceding message from A has been stably received. This delay is denoted by the arc. An application that continuously alternates communication between two groups by sending messages asynchronously, will in fact see no advantage to the asynchronous call, since each communication context switch will essentially force synchrony on the previous message send.

3.3 Named groups

Previous implementations of ISIS have incorporated the naming service into the same server process that manages the group membership protocols. This process, historically known as *protos* (for protocol server), resides on every ISIS site. (For scaling reasons ISIS V3.0 allows for remote connections that are less fault-tolerant and do not run the protocol server directly but instead connect to a "mother" ISIS site.) The implementation of the name service ensures one-copy consistency of the name space mappings among all of the protos processes. This has a great impact on the cost of creating a named group as indicated in the transaction tool discussion above.

4 Overall Design

The following observations about the common uses of process groups guided us in our design to combat these problems. We have found that many applications use

- many process groups.
- heavily overlapping groups.
- both small groups and large groups.
- unnamed groups.

With the number of groups far exceeding the number of processes in the system, high overlap and coincidence of groups is unavoidable. We observed that by combining overlapping process groups so that they share a single "core" process group, we could obtain several distinct advantages. A careful look at the performance problems shown above revealed that for the common case of identical overlapping groups, the protocols being exercised were largely unnecessary. Consider the group membership protocol in the case of process failures: if a single core process group were used instead of a thousand identical groups, only a single flush would be necessary to ensure failure atomicity and instantiate the new group view. Similarly, using only a few core groups can reduce transmission delays (for obtaining stability) and thus increase truly asynchronous message sends. Much of the state maintained by the ISIS transport system to maintain causality and other ordering semantics can be shared by these light-weight groups, reducing the resource requirements of the system.

Thus there is much to be gained by separating the protocols underlying the process group implementation from the interface provided to the applications programmer. As was the solution in the above distributed systems examples, we manage a large collection of light-weight process groups by mapping them onto relatively small sets of "core" process groups. These core groups are the groups provided by the VSync (for virtually synchronous) kernel in Figure 3.

Figure 3 shows the architecture of HORUS. A goal of HORUS is to take advantage of the microkernel architectures being offered by modern operating systems. Our experience with the ISIS system has allowed us to reorganize the major components of the system in a layered and modular fashion, suitable for use in microkernels. The lowest layer of HORUS called MUTS (MUlticast Transport Service) [16, 17], provides a portable abstraction of the underlying operating system to the higher layers. The operating system specific code is isolated in configuration dependent source files within MUTS. This foundation allows for easy porting of the system to operating systems such as Mach [1], Chorus [4, 5], and Amoeba[11]. A key component of MUTS is the abstraction it provides of a multicast transport service. MUTS isolates the higher layers from the details of underlying transport protocols, yet provides important feedback information to the higher layers so that they may deal with communication failures in a consistent, well-defined manner. Above MUTS, the VSync kernel provides ordering semantics on multicasts, and provides the basic process group abstraction with strong semantics on the ordering of group events with respect to multicasts. These two layers define the portion of the architecture that is appropriate to put in the system space of an operating system. While this is not necessary, it will likely yield more efficient communication. The layers above this are most appropriately placed in a user space library. This is where the light-weight process group subsystem lives. The subsystem provides an interface to applications through this library and is used by the other tools within the library itself. The library also contains tools for managing replicated data and distributed computations.

5 Design Issues

In this section we examine a number of the issues which we faced during the design of the LWG subsystem. We wanted a flexible, efficient, portable, and simple interface to the subsystem. The interface had to allow for tight control of the light-weight to core group mapping for use as a research tool and by sophisticated users, yet also allow the subsystem itself to manage this mapping in an intelligent way for ordinary users of the system. Efficiency was paramount; to be useful, the system had to optimize the critical path. In the next few sections we discuss the major issues in designing the LWG subsystem.

5.1 Mapping LWGs to core groups

To address the goals of flexibility and simplicity we introduced the notion of *core group sets* which can be managed by the subsystem or the user. A core group set is a collection of ISIS process groups which are used as the communication transports for light-weight groups. Light-weight groups are allocated out of a core group set and are always mapped to exactly one core group in the set. Influenced by the Mach [1] philosophy of separating policy from mechanism we provide default routines to manipulate these sets together with hooks in the interface where the user can have tight control of the mapping between a light-weight group and its core group. The default policy manages core groups completely within the LWG subsystem. In this case the subsystem creates, changes, and deletes core groups in the set dynamically as the mapping needs of the LWGs change over time and uses heuristics to define the mapping.

Core group sets allow us to address several issues at once. First, they provide flexibility. By providing support for multiple sets, varying levels of mapping control may be used within the same application. This allows different mapping policies to be enforced for different types of objects. For example, one policy might mandate that the membership of a light-weight group exactly match the membership of its core group, while another might allow LWG members to be a subset of the members of the core group. These policies will have different impacts on the performance of the system. Second, by providing policies for self-management together with a default core set, the system provides much of the functionality of light-weight groups with a simple interface. Third, by constraining LWGs to map only to those core groups within their core group set, we improve the efficiency of self-management policies by reducing the search space for core groups.

In Figure 4 we show a mapping of 3 different light-weight groups onto a common core group. It is important to note that the membership of the core group need not match the membership of the light-weight group exactly; it can be larger. However, there are tradeoffs with such mappings. If hardware multicast is not available, the cost of sending a multicast message may be greater due to the increased number of recipients. In Figure 2 (b) we see that processes *A* and *D* receive extra messages which the light-weight group subsystem will need to filter out. However, these extra messages must be weighed against the acknowledgement and stability information messages sent in diagram 2(a). If hardware multicast is in use, the extra members do not add to the cost of sending a message, but the extra members themselves still pay a cost for handling the receipt of the message. On the other hand, supporting "subset mappings" yields a number of advantages. First, the number of core groups that are needed is reduced since they can encompass more light-weight groups. This reduces the amount of state that is needed to support causality, reduces the number of communication context switches that occur, and reduces the size of the space that must be searched when creating a new mapping for a LWG. Second, with fewer core groups, better use can be made of hardware multicast addresses. This can be a critical performance factor since hardware devices such as Ethernet interfaces support a fairly limited number of multicast addresses before they go into "software" mode. Third, the cost of adding a member already in the core group to the LWG is cheaper since much of the state of the member has already been set up by the core group.

Over time core groups will have a number of different LWGs mapped to them and at some point a core group may have no LWGs that map to it. To avoid consuming too much memory, such core groups have to be garbage collected periodically. This collection could occur at the instant the set of mapped LWGs becomes empty, but leaving the core group around for some grace period can be advantageous in the event that a subsequent LWG mapping appears soon. In the transaction tool this does well on the common scenario where a client issues a series of transactions to the same set of servers, when the grace period is longer than the time between transactions. Thus we could exploit temporal, as well as spatial, locality of transactions.

Under high load the LWG subsystem can be faced with a potentially large search problem. Upon the creation of a LWG with an initial set of members, it must map this group to an existing core group, if possible. Determining the best mapping can, without using good search techniques, lead to a linear search of the core group set, which in the worst case can be quite expensive (for n processes, there are potentially $2^n - 1$ unique core groups). In practice such a large number of core groups never exists since the presence of subset mappings eliminates the need for many of these groups. In any case, the default mapping policy of the LWG subsystem manages this search by using a hash index scheme keyed on the *membership* of the group. This enables the search to quickly narrow in on a core group containing the right members. The policy of this default is to map an LWG l to the smallest core group G whose membership contains the processes in l. A further constraint is that the number of extra members in G must not exceed some threshhold k (a parameter of the heuristic). If no such group is found, a new group is created with the membership of l. Our performance measurements used this heuristic with k set to $MIN(5, |l| * 2)$ where $|l|$ is the number of members in l, and show that even this simple approach works well. We are currently experimenting with other heuristics.

5.2 Added Functionality

Rewriting ISIS gives us the opportunity to consider providing different forms of group semantics. ISIS provides a broad range of ordering semantics for its communication (MBCAST, FBCAST, CBCAST, ABCAST, and GBCAST)[9], yet only one set of semantics is provided for the process group mechanism. While it can rightfully be argued that too many choices only leads to the confusion of the programmer, it is nonetheless interesting to consider the use of this subsystem as tool for research into a spectrum of process group semantics. An example clearly establishes the validity of this argument. We have observed that while many applications benefit greatly from the strong semantics of ISIS process groups, there are nonetheless a number of applications for which these semantics are too strong and which would benefit from the performance improvements obtained by using weaker semantics. Consider a collection of sensor processes responsible for periodically sensing the temperature of a room and reporting on these values to a collection of reader processes. For fault-tolerance multiple sensors are used, and the reader processes collect the sensor data to determine an average for the room's temperature. Here an ISIS process group may be used as the group communication transport. The sensor processes would, on initialization, join the group and start broadcasting data. Notice, however, that the sensors themselves use the group for sending only; they do not need to obtain state from other members and are not concerned about the order in which they join the group. In this situation ISIS would completely order the joins when in fact this is not needed.

5.3 Large numbers of process groups

Just as light-weight threads share their state within the address space of their encompassing process, light-weight groups share their causality context and group data structures within their core group. The reduced memory resource needs combined with the sharing of the core group protocols for failure detection and causality allow HORUS to efficiently support many more light-weight process groups than core groups.

6 Interface

Table 1 shows the interface to the light-weight group subsystem. This interface provides asynchronous results to enable the application to take advantage of pipelining to improve its efficiency and yet retain a simple model of execution.

7 Initial Performance Results

As a proof of concept, we built a prototype of the light-weight group subsystem on top of ISIS V3.0.6. Doing so allowed us to proceed with our research testing in parallel with the building of the HORUS system, which is being built bottom up. The lowest layers of HORUS are almost now complete and the building of the light-weight group subsystem on top of HORUS is just beginning. Building the prototype on top of ISIS V3.0.6 allows us to make measurements of the impact of the LWG subsystem on the performance of the system. Happily, the prototype

Function	*Arguments*	*Result*	*Description*
lwg_create	initial members	lwg	Create light-weight group.
lwg_add	members	—	Add members to a group.
lwg_remove	members	—	Remove members.
lwg_destroy	lwg	—	Destroy group.
lwg_send	lwg, msg	send_id	Post message to a group.
lwg_receive	lwg	msg, recv_id	Wait for next message.
lwg_reply	recv_id, reply msg	—	Send a reply.
lwg_get_next_reply	send_id	msg	Wait for next reply.
lwg_discard_replies	send_id	—	No more replies wanted.

Table 1: The light-weight group interface.

showed significant improvements in performance and the results supported our initial suspicions.

Initial measurements of the performance of our light-weight process group subsystem are encouraging. The following measurements were taken on Sun 4c/60 Sparc 1+ workstations running Sun OS 4.1.1 using ISIS V3.0.6.

Our measurements of the cost of obtaining message stability confirmed our initial expectations. Switching communication from one core group to another core group costs the application approximately one synchronous multicast. For applications that change contexts frequently with respect to message sends, this overhead can be significant. For example, a process that repeatedly switches between to coincident core groups runs roughly twice as long as the equivalent program sending to only one core group. Asynchronously CBCASTing 400 byte messages to 4 members (3 remote, 1 local) costs 18.0 ms per multicast in the strictly alternating case, and only 10.4 ms in the single group streaming case. For 2 members (1 remote, 1 local), the cost of alternating CBCASTs is 10 ms, for streaming it is 3.2 ms. The tuning of the transport layer plays an important factor in the cost of obtaining stability. For efficiency the transport layer will attempt to determine if the sending application is in a streaming or "interactive" mode. In the former, the transport layer will delay acknowledgements in order to send as few ack messages as possible, in the latter case the transport layer is aggressive about sending acks, so that the cost of the context switch is as small as possible.

To measure the effect of light-weight groups on reducing the costs of a join, we compared creating bursts of 100 LWGs vs. core groups. The prototype LWG subsystem makes use of a group view manager which replaces the role of "protos" for managing views and group names. We ran these tests with the creating process both local and remote to the view manager. In the local case, a LWG create took 45 ms compared to 60 ms. In the remote case, a LWG create took 29 ms compared to 200 ms for the core group. Contention for the processor may partially explain why the LWG create with the local view manager is more expensive than the remote case, but this is still curious. These results are preliminary and only serve as proof of concept. The LWG subsystem on HORUS will not use a group view manager and will use a separate name service for named groups.

We measured the time of a light-weight group leave event for both the local and remote view manager cases. Under both situations the cost of a light-weight group leave was 9 ms. The cost of a core group leave for the remote case was 197 ms, and for the local leave it was 80 ms.

8 Related Work

The Transis system [3, 2] provides *process sets* at the session layer of their system. These are closely related to the multiplexing layer of HORUS. Their job is to map a process abstraction of membership onto a site abstraction of membership. Transis has a single "Lansis" process on each processor which coordinates the current configuration set (CCS), the set of currently active processors. In contrast with ISIS and Horus, Transis has a single CCS within a *broadcast domain* (typically a LAN). All of the processors within this CCS receive all messages sent within any process set in that CCS. Transis uses local broadcasts to reduce the impact of this, but this can have an adverse affect on active processors that are not interested in participating in the broadcast domain.

The ESPRIT Delta-4 project [12] also provides a similar light-weight notion of process groups. They provide a sub-grouping mechanism in their extended atomic multicast protocol service (xAMp). This yields support for multiple selective address lists which share the context of a *gate group*. In many respects these are similar to process lists in ISIS [9]. Both of these mechanisms however require the user to determine the mapping between the address list and the group. Once defined this association is permanent. The address lists are purely local to the creating process, indeed the members may not even be aware of their membership in a list. These are significant differences between the light-weight group mechanism reported here and process lists.

The Delta-4 project has also recognized the importance of providing different qualities of service within their group communication service. This recognition lead in part to the evolution of the Delta-4 AMp service to xAMp. Our observation of this need has been similar in the ISIS system.

9 Conclusion

It is interesting to draw analogies with the evolution of some other common system paradigms. Memory allocation is an excellent example. Before the advent of standard library routines like malloc, programmers were forced to implement their own memory allocator routines which usually had the effect of reducing the portability of their software, since their memory allocators were often OS and machine specific. Today, malloc is widely available, and the mechanisms by which memory is allocated are hardly a concern to most programmers. Much like malloc, light-weight process groups abstract away the details of the implementation, yet provide added functionality and improved performance.

Similarly, threads have become an attractive mechanism for improving the performance of processes. Threads reduce the heavy-weight context switching of processes by sharing an address space among the threads of control. The sharing of resources seems to be a common theme to providing light-weight mechanisms. We are encouraged by the initial results of our prototype and are actively incorporating these ideas into HORUS.

Currently, we are actively experimenting with prototype and are building the light-weight process subsystem and user-level libraries on top of the VSync kernel in HORUS. We hope to have a release of this system available by the end of 1993.

10 Acknowledgements

The authors of this paper gratefully acknowledge the contributions of Sue Honig, Keith Marzullo, Aleta Ricciardi, Alex Siegel, Patrick Stephenson, and Mark Wood to ISIS and its applications. We also greatly appreciate the helpful comments of Cliff Krumvieda and Mike Reiter on an earlier draft of this paper.

References

[1] Mike Accetta, Robert Baron, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for UNIX development. In *Proceedings of the USENIX Summer '86 Conference*, pages 93 - 112, Atlanta, GA, June 1986.

[2] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Membership algorithms in broadcast domains. Technical Report CS92-10, The Hebrew University of Jerusalem, June 1992.

[3] Yair Amir, Danny Dolev, Shlomo Kramer, and Dalia Malki. Transis: A communication sub-system for high availability. In *Proceedings of the Twenty-Second International Symposium on Fault-Tolerant Computing*, pages 76–84, Boston, Massachusetts, July 1992. institution of Electrical and Electronic Engineers.

[4] François Armand, Michel Gien, Frédéric Herrmann, and Marc Rozier. Revolution 89 or "Distributing UNIX Brings it Back to its Original Virtues". Technical Report CS/TR-89-36.1, Chorus Systèmes, 6, avenue Gustave Eiffel, F-78182, Saint-Quentin-En-Yvelines, France, October 1989.

[5] François Armand, Frédéric Herrmann, Jim Lipkis, and Marc Rozier. Multithreaded Processes in Chorus/MIX. Technical Report CS/TR-89-37.3, Chorus Systèmes, 6, avenue Gustave Eiffel, F-78182, Saint-Quentin-En-Yvelines, France, October 1989.

[6] Ken Birman, Andre Schiper, and Pat Stephenson. Lightweight causal and atomic group multicast. *Transactions on Computer Systems*, pages 272–314, August 1991.

[7] Kenneth P. Birman. The process group approach to reliable distributed computing. Submitted to the Communications of the ACM.

[8] D. R. Cheriton and W. Z. Zwaenepoel. Thoth, a portable real-time operating system. *Communications of the Association for Computing Machinery*, pages 105 - 115, February 1979.

[9] The ISIS Group. *The ISIS Distributed Toolkit Version 3.0 User Reference Manual*. Department of Computer Science, Cornell University, 3.0 edition, May 1991.

[10] Keith Marzullo, Robert Cooper, Mark Wood, and Ken Birman. Tools for distributed application management. *Computer*, 24(8):42–51, August 1991.

[11] S. J. Mullender and A. S. Tanenbaum. The design of a capability-based operating system. *The Computer Journal*, 29(4):289–300, March 1986.

[12] D. Powell, editor. *Delta-4: A Generic Architecture for Dependable Distributed Computing*, volume 1. Springer-Verlag, 1991.

[13] A. M. Ricciardi and K. P. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Procedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 341–353. ACM, August 19-21 1991.

[14] Alexander Siegel. *Performance in Flexible Distributed File Systems*. PhD thesis, Cornell University, February 1992. PhD Thesis, 92-1266.

[15] Avadis Tevanian, Jr., Richard F. Rashid, David B. Golub, David L. Black, Eric Cooper, and Michael W. Young. Mach threads and the Unix kernel: The battle for control. Technical Report CMU-CS-87-149, Carnegie-Mellon University, August 1987.

[16] Robbert van Renesse, Kenneth Birman, Robert Cooper, Bradford Glade, and Patrick Stephenson. Reliable multicast between microkernels. In *Proc. of the USENIX workshop on Micro-Kernels and Other Kernel Architectures*, pages 269–283, April 1992.

[17] Robbert van Renesse, Robert Cooper, Bradford Glade, and Patrick Stephenson. A RISC approach to process groups. In *Proc. of the 5th ACM SIGOPS Workshop*. IRISA INRIA, September 1992.

[18] Mark Wood. *Fault-Tolerant Management of Distributed Applications Using the Reactive System Architecture*. PhD thesis, Cornell University, December 1991. PhD Thesis 91-1252.

11 Figures

Figure captions:
Caption for figure 1.

Diagrams (a) and (b) show four processes, A-D, joined to a single process group, denoted by the encompassing oval. C crashes around the same time that A sends a message to the group. (a) shows multicast communication that does not respect failure atomicity; B and D receive the message in different views of the group. The multicast in (b) respects failure atomicity.

Caption for figure 2.

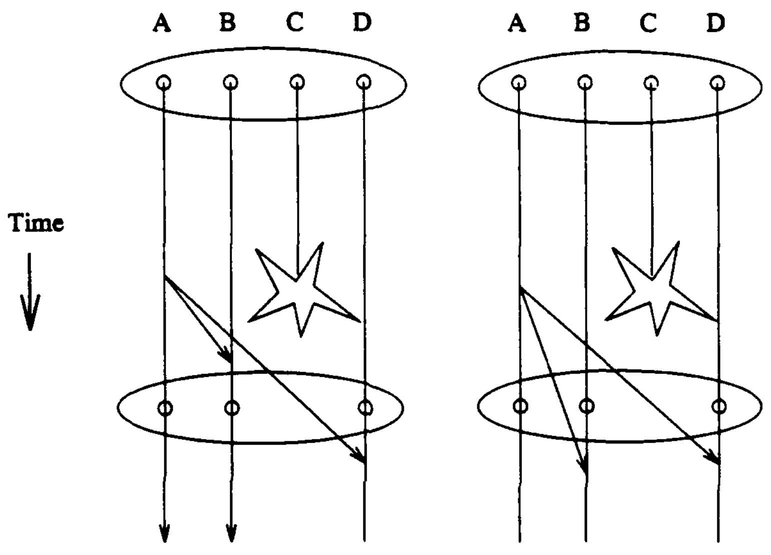
Diagram (a) shows two process groups (represented by ovals) and the messages sent by the system during when communication switches from group A,B,C to group B,C,D. The solid arrows represent the application multicasts, the dashed arrows represent low-level acknowledgements, and the dotted arrows represent messages containing message stability information. Diagram (b) shows the message traffic for the same pair of application multicasts, but with the two groups merged into one. The arced arrows represent delayed messages, in (a) by the sender, and in (b) by the receiver.

Caption for figure 3

The HORUS Architecture.

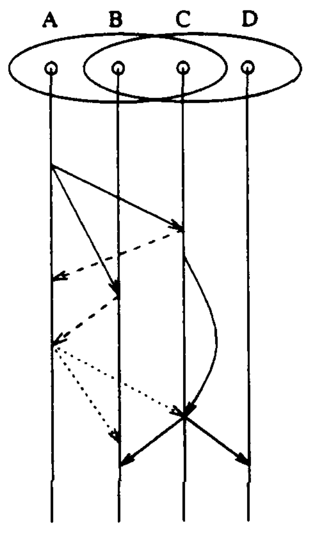
Caption for Figure 4

A mapping of three light-weight groups onto a common core group.

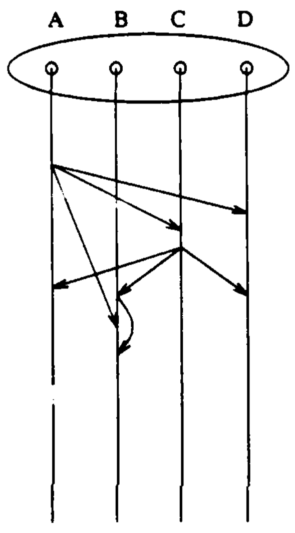


(a) Without Failure Atomicity (b) With Failure Atomicity

Figure 1:



(a)



(b)

Figure 2:

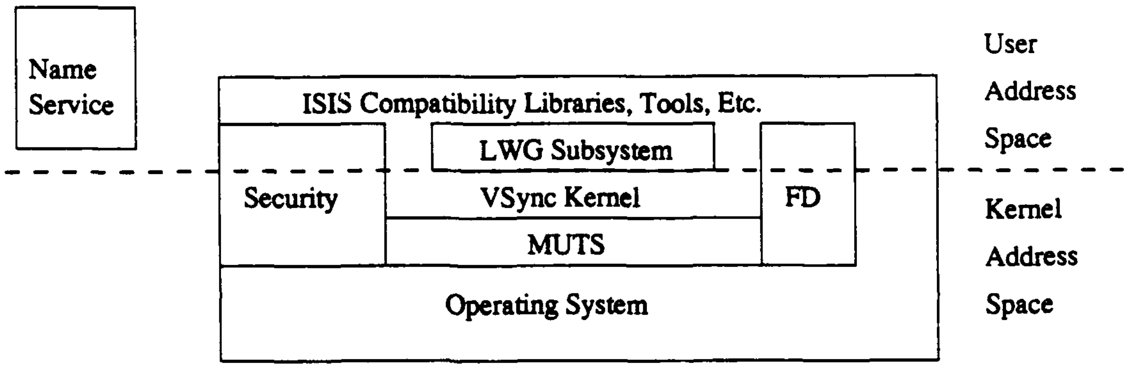

Name Service
ISIS Compatibility Libraries, Tools, Etc.
LWG Subsystem
Security
VSync Kernel
FD
MUTS
Operating System
User Address Space
Kernel Address Space

Figure 3:

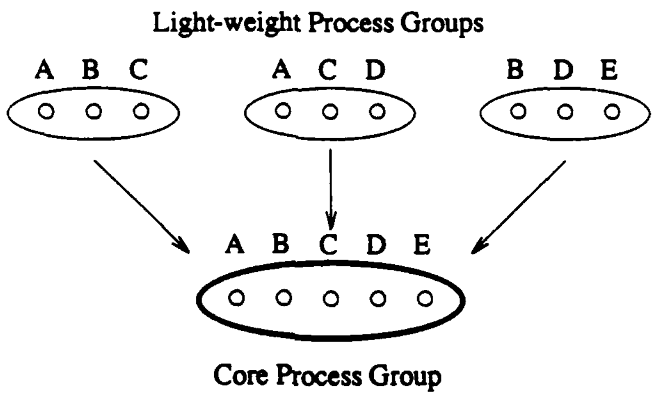

Light-weight Process Groups
A B C
A C D
B D E
A B C D E
Core Process Group

Figure 4: